852L1-DataCoding-Lecture10

December 11, 2020

1 Lecture 10: Data Visualisation with Python

In our previous lecture, we introduced the basics of data analysis with Python using the Pandas library.

In this lecture, we will cover some of the visualisation tools available in Python and the main library for plotting and creating graphs, i.e. **Matplotlib**.

1.1 Introductory example: unemployment rates

Let's start with a simple, but rich enough, example that illustrates some of the basic features of plotting in Python. We will then cover each of the features in detail in order to understand how they work.

Our goal for this example is to create a plot showing how the unemployment rate in different U.S. regions evolves over time. First, let's create a few variables that will represent our data.

[1]:
```
# Create a dictionary of lists containing unemployment rate for each U.S. region
unemployment = {
    "NorthEast": [5.9,  5.6,  4.4,  3.8,  5.8,  4.9,  4.3,  7.1,  8.3,  7.9,  5.
↪7],
    "MidWest": [4.5,  4.3,  3.6,  4. ,  5.7,  5.7,  4.9,  8.1,  8.7,  7.4,  5.
↪1],
    "South": [5.3,  5.2,  4.2,  4. ,  5.7,  5.2,  4.3,  7.6,  9.1,  7.4,  5.5],
    "West": [6.6, 6., 5.2, 4.6, 6.5, 5.5, 4.5, 8.6, 10.7, 8.5, 6.1],
    "National": [5.6, 5.3, 4.3, 4.2, 5.8, 5.3, 4.6, 7.8, 9.1, 8., 5.7]
}

# Create a list that represents the years
years = list(range(1995, 2017, 2))
```

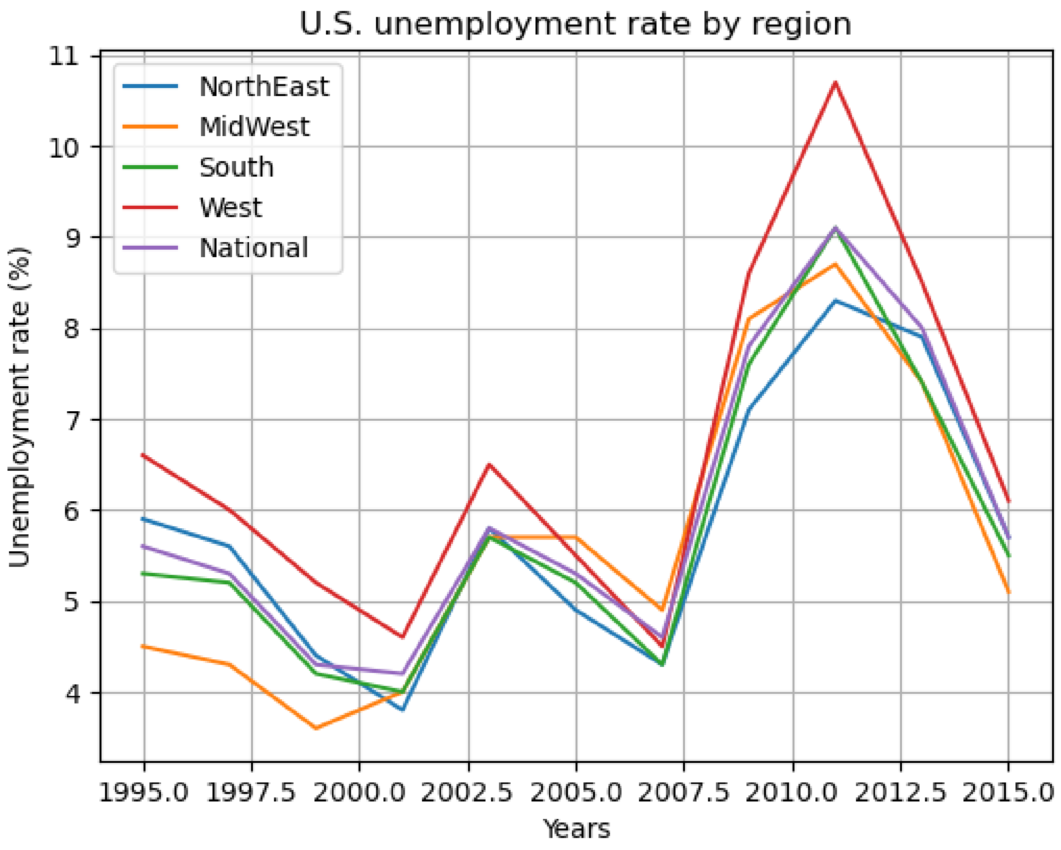
Given the unemployment data stored in a dictionary and the associated years in a list, we can create a time-series plot in two steps:

1. Import the plotting module **pyplot** from the library **matplotlib**.
2. Create and show the plot.

[2]:
```
# Import the plotting library
import matplotlib.pyplot as plt
plt.style.use("default")
```

Now that Python has access to the necessary plotting functions, we are ready to create the graph with the following code

[3]:
```python
# Sequentially plot unemployment rate on the same figure
for region, unemp_rate in unemployment.items():
    plt.plot(years, unemp_rate, label=region)

# Display useful information
plt.title("U.S. unemployment rate by region")
plt.xlabel("Years")
plt.ylabel("Unemployment rate (%)")
plt.grid()
plt.legend()
plt.show()
```



Even a simple graph like the one above illustrates the advantages of data visualisation. The time-series plot immediately shows in which years the unemployment rate was higher than average, and in which region the unemployment rate was the highest.

In order to create the plot, we had to import the **Matplotlib** library first. Any time we are working with plots and other visualisation tools, we can import the `pyplot` module of Matplotlib with

```
import matplotlib.pyplot as plt
```

2 Matplotlib: the plotting library

Matplotlib is the de facto standard library for plotting in Python.

According to the official website:

> **Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Matplotlib is a very powerful and flexible library that allows researchers and practitioners to produce publication quality plots with a high degree of customisation.

Matplotlib is also used internally by other libraries as their main plotting engine. Examples are the data analysis library Pandas and the data visualisation library **Seaborn**.

In what follows, we will go over the main features and functionalities of Matplotlib, starting from plotting a simple line.

2.1 A simple (x, y) plot

Let's say that we want to plot a straight line. Mathematically, a straight line is represented by the linear relationship

$$y = ax$$

where x are all real numbers measured on the horizontal axis, a is a constant, and y are the correspondent numbers measured on the vertical axis.

To understand how Matplotlib works, it is best to view this linear relationship as a **sequence of points in the (x, y) plane**. For instance, given $a = 2$, the linear relationship for the first 5 natural numbers is summarised by

x	1	2	3	4	5
y	2	4	6	8	10

In order to create a line, the function `plt.plot()` takes **a sequence of x values** and **a sequence of associated y values**, and maps the x and y values to the correspondent (x, y) points on the (x, y) plane.

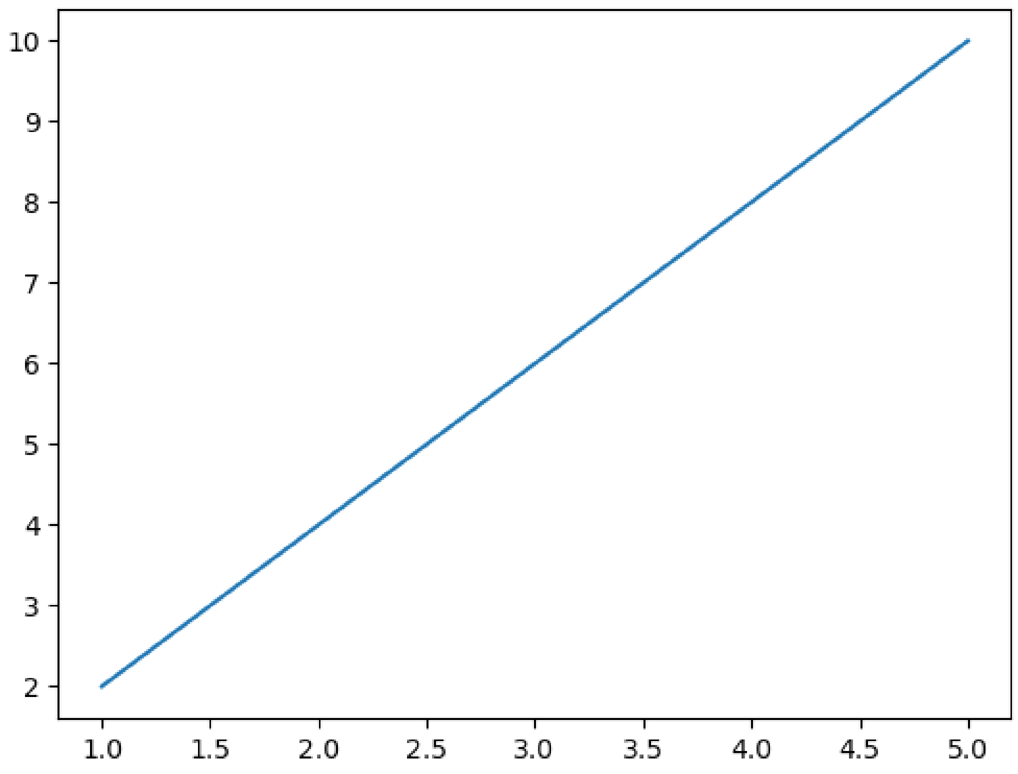
In its most basic form, the syntax for creating a line plot is

```
plt.plot([sequence of x values], [sequence of y values])
```

Therefore, the linear relationship $y = 2x$ is easily plotted as follows

[4]:
```
# Create two lists of x and y coordinates
x_values = [1, 2, 3, 4, 5]
y_values = [2, 4, 6, 8, 10]
```

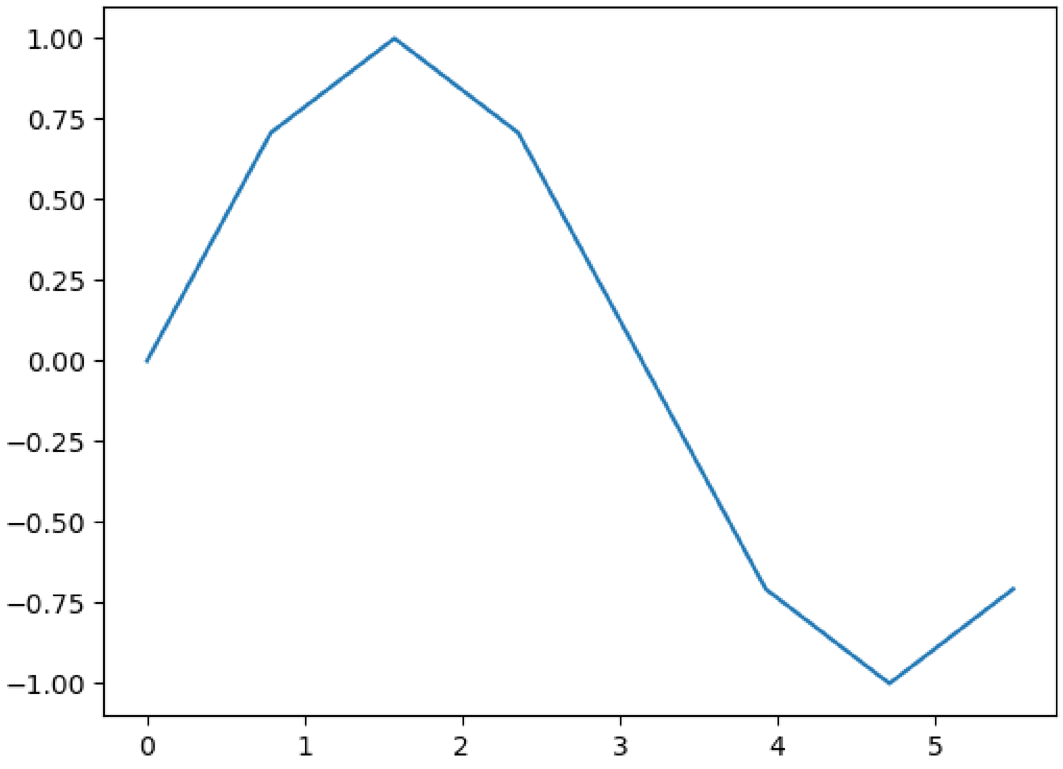
```
# Create the plot with plt.plot()
plt.plot(x_values, y_values)

# Display the plot
plt.show()
```



Of course, the same procedure applies to more complicated functions. For instance, if we want to plot the trigonometric function $y = \sin x$ between 0 and 2π we can do that with

[5]:
```
# Import sin() function and pi constant from math library
from math import sin, pi
# Import arange() function from numpy to create a range of floats
from numpy import arange

# Create x and y=sin(x) coordinates
x_values = arange(0, 2*pi, pi/4)
y_values = [sin(x) for x in x_values]

# Create the plot with plt.plot()
plt.plot(x_values, y_values)

# Display the plot
```

```
plt.show()
```



2.2 Plotting multiple lines

There are many situations in which we want to have two or more lines in the same graph. This is very useful when we want to plot two or more mathematical functions, or compare the behaviour over time of two or more variables. The unemployment example above is one of such cases.

Plotting multiple lines is easily done by sequentially calling the `plt.plot()` function, each time with a different set of (x, y) coordinates. That is

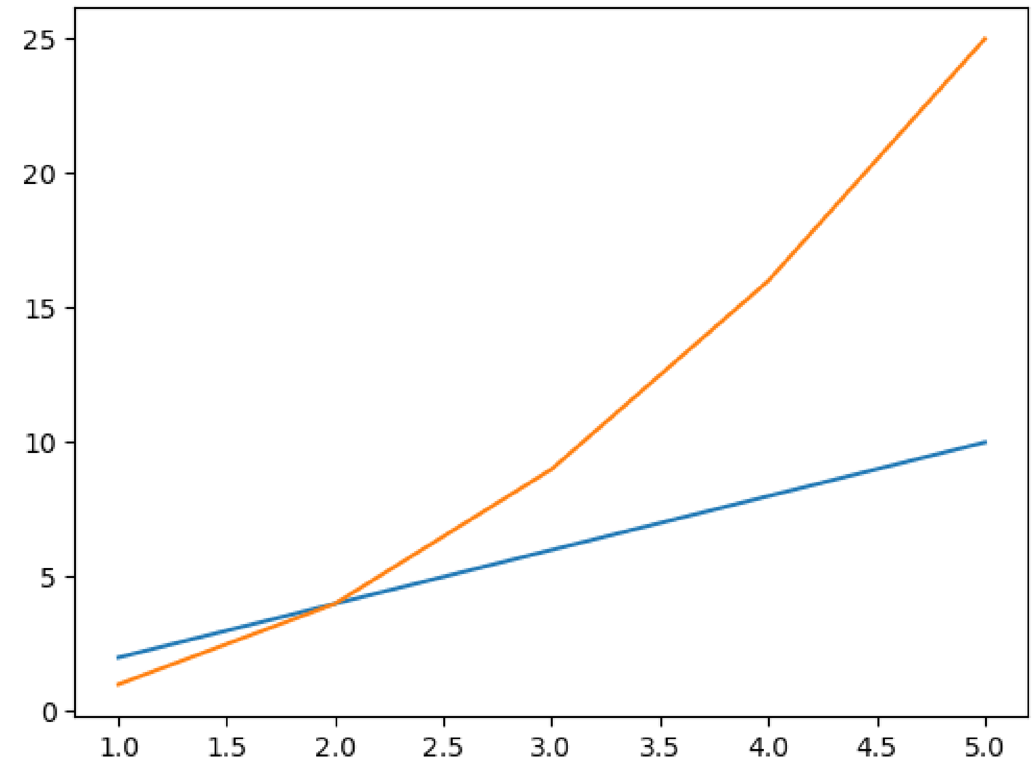
```
plt.plot([1st sequence of x values], [1st sequence of y values])
plt.plot([2nd sequence of x values], [2nd sequence of y values])
...
plt.plot([nth sequence of x values], [nth sequence of y values])
```

In the background, Matplotlib creates a new **figure** on the first call of `plt.plot()` and then subsequently adds new lines to the **same** figure on subsequent calls of `plt.plot()`.

Note: In a Jupyter notebook, subsequent calls of `plt.plot()` must be in the same cell in order to produce the desired result.

```
[6]: # Create (x, y) coordinates representing y=2x
x1 = [1, 2, 3, 4, 5]
```

```
y1 = [2, 4, 6, 8, 10]

# Create the first plot
plt.plot(x1, y1)

# Create (x,y) coordinates where y is the square of x
x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]

# Create the second plot
plt.plot(x2, y2)

# Display the graph
plt.show()
```



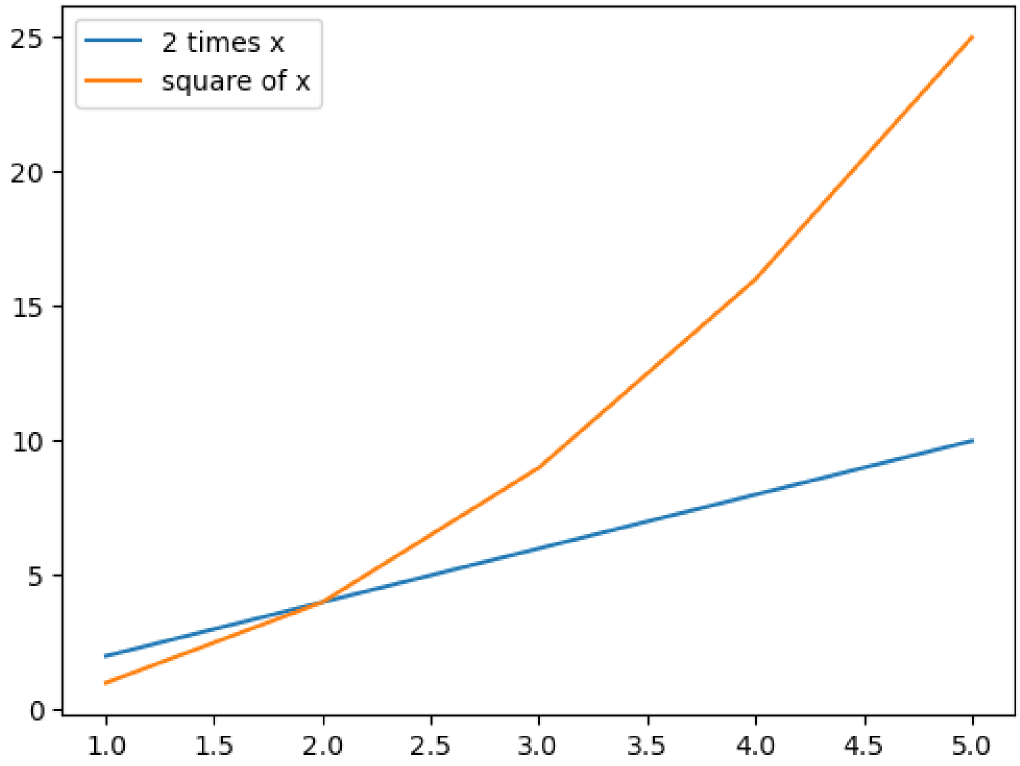
2.3 Adding information to your graph

We have now created a simple plot with two lines but we are far from having a complete graph. Every time we want to include a graph in a report or in a presentation, the graph should display a minimum amount of information to make the reader aware of what the graph is showing.

2.3.1 Adding a legend

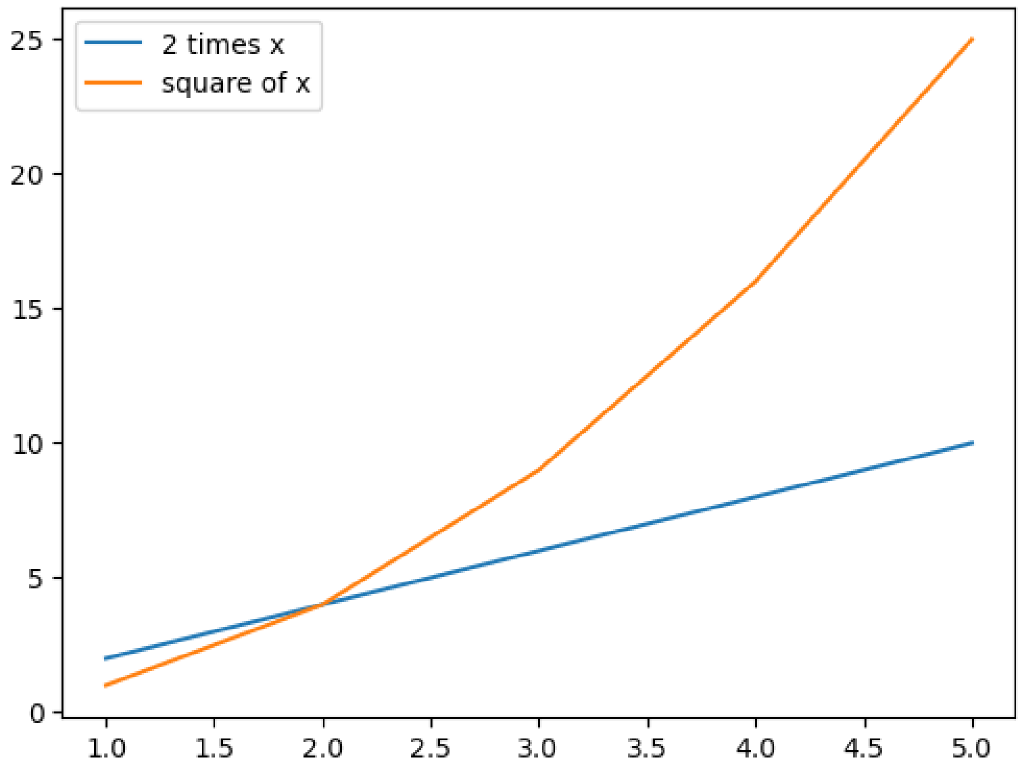
Whenever we are plotting multiple lines in the same figure, it is a really good idea to add labels that indicate what each line represents. This is done by creating a **legend** via the `plt.legend()` function.

[7]:
```
x1 = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
plt.plot(x1, y1)

x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]
plt.plot(x2, y2)

# Create the legend with a list of string as argument
plt.legend(["2 times x", "square of x"])

# Display the graph
plt.show()
```



Or alternatively

[8]:
```python
x1 = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]

# Add a keyword argument label along with the desired string
plt.plot(x1, y1, label="2 times x")

x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]

# Add a keyword argument label along with the  string
plt.plot(x2, y2, label="square of x")

# Create the legend
plt.legend()

# Display the graph
plt.show()
```



2.3.2 Adding axes labels

We can indicate what the unit of measurement are on x and y axes with the `plt.xlabel(label_string)` and `plt.ylabel(label_string)` functions, respectively. Both functions will take a string as an argument.

### Adding a title

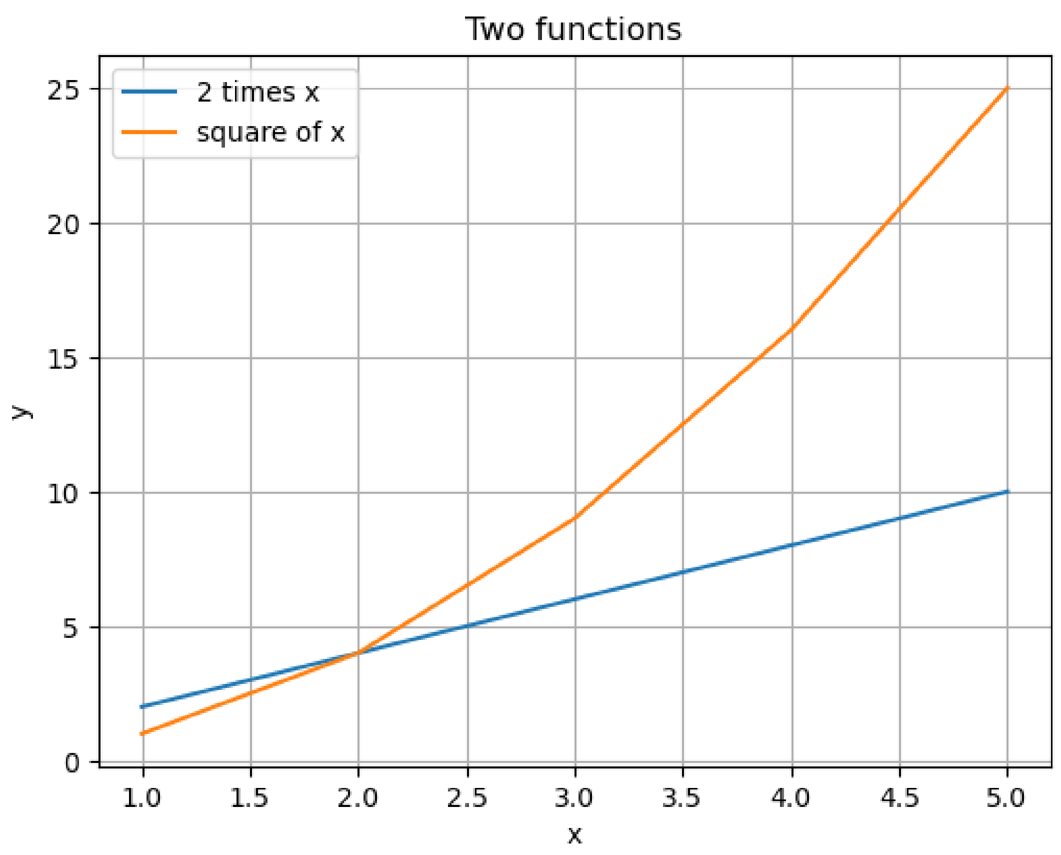
A figure title is added with `plt.title(title_string)` where a string should passed as an argument.

2.3.3 Adding a grid

Optionally, a grid can be added to increase the readability of the graph.

Let's now add all these elements to our simple graph with two lines:

[9]:
```
x1 = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
plt.plot(x1, y1, label="2 times x")

x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]
plt.plot(x2, y2, label="square of x")

# Create the legend
plt.legend()

# Add axes labels
plt.xlabel("x")
plt.ylabel("y")

# Add title
plt.title("Two functions")

# Add grid
plt.grid()

# Display the graph
plt.show()
```



2.4 Plot customisation

A nice feature of Matplotlib plots is that you can change the default appearance of the graph to one of your choice.

### Customising lines

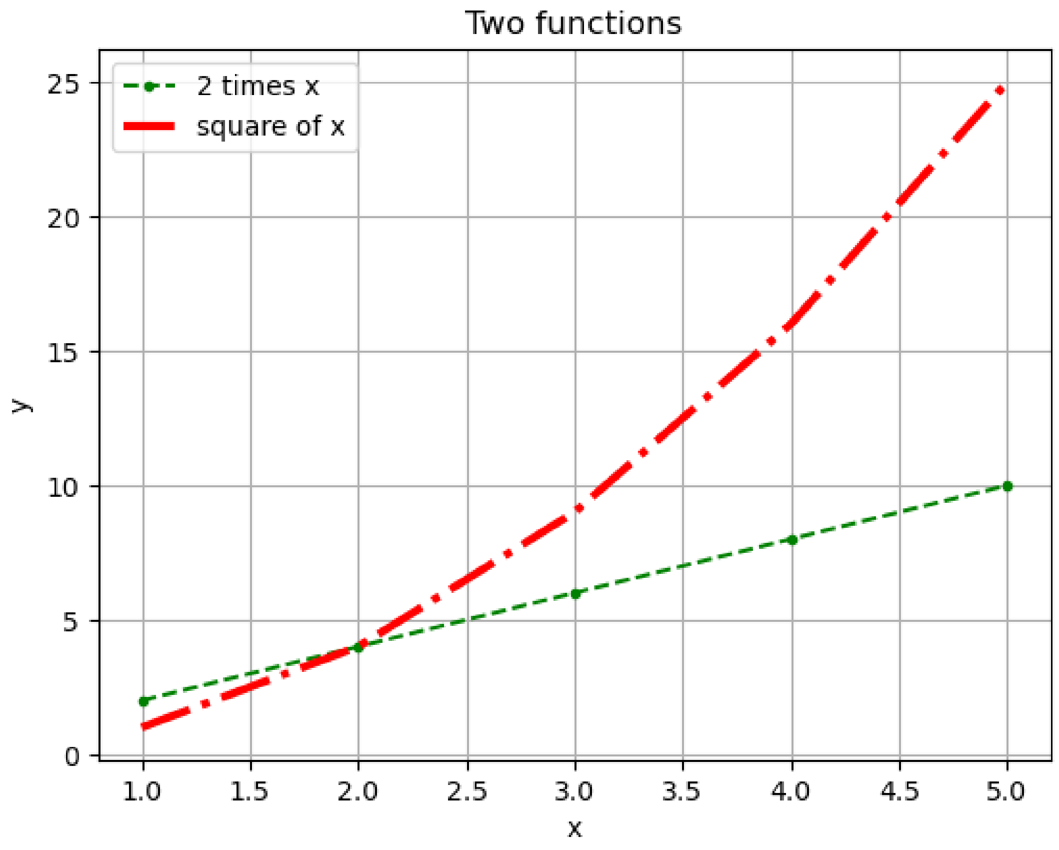
Individual lines in a graph can be customised in many different ways by passing a comma-separated list of keyword arguments to `plt.plot()` when the plot is created.

Here are some of the possible customisation options:

- `color`: Sets the color of the line
  - Example values: {'b' for blue, 'g' for green, 'r' for red}
  - Reference page with all options here
- `linestyle`: Set the style of the line
  - Values: {'-' or 'solid', '–' or 'dashed', '-.' or 'dashdot', ':' or 'dotted'}
- `linewidth`: Set the thickness of the line
  - Values: numerical (float)
- `marker`: Set whether to display a marker at (x, y) coordinate
  - Example values: {'.' for point, 'o' for circle, 's' for square}
  - Reference page with all options here

Let's use the graph above to show a customisation example

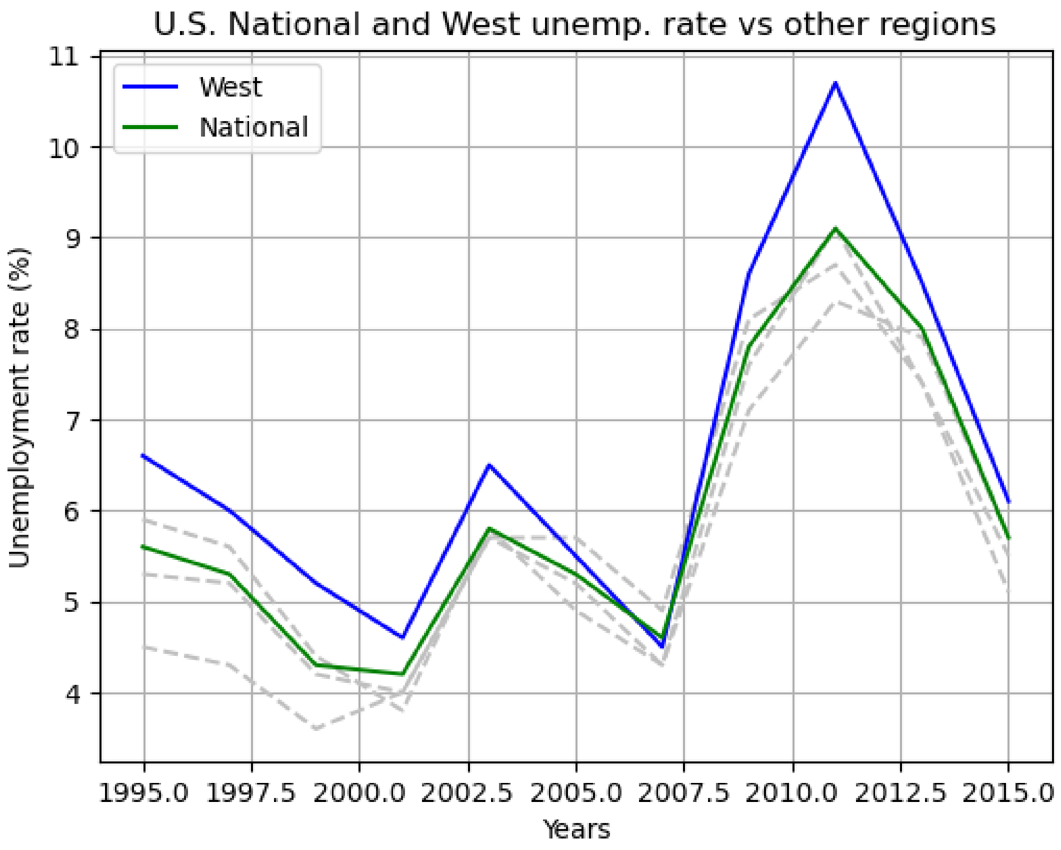
[10]:
```
x1 = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]

# Customise the line appearance
plt.plot(x1, y1, color="g", linestyle="--", marker=".")

x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]

# Customise the line appearance
plt.plot(x2, y2, color="r", linestyle="-.", linewidth=3)

# Add info
plt.legend(["2 times x", "square of x"])
plt.xlabel("x")
plt.ylabel("y")
plt.title("Two functions")
plt.grid()

# Display the graph
plt.show()
```



Line customisation can be really useful if we want to highlight specific features of the data.

Using the U.S. unemployment data, suppose that we want to highlight the National and West unemployment rates. The goal is to create a graph where the evolution of these variables stand out immediately.

[11]:
```
# Highlight the unemployment rate of National and West
for region, unemp_rate in unemployment.items():
    if region == "National":
        plt.plot(years, unemp_rate, label=region, color='g')
    elif region == "West":
        plt.plot(years, unemp_rate, label=region, color='b')
    else:
        plt.plot(years, unemp_rate, linestyle="--", color='0.75')

# Display useful information
plt.title("U.S. National and West unemp. rate vs other regions")
plt.xlabel("Years")
plt.ylabel("Unemployment rate (%)")
plt.grid()
plt.legend()
plt.show()
```



2.4.1 Customisation styles

In addition to customise individual lines, you have the option of changing the overall appearance of the entire graph with `plt.style.use()`.

You can check with styles are available with

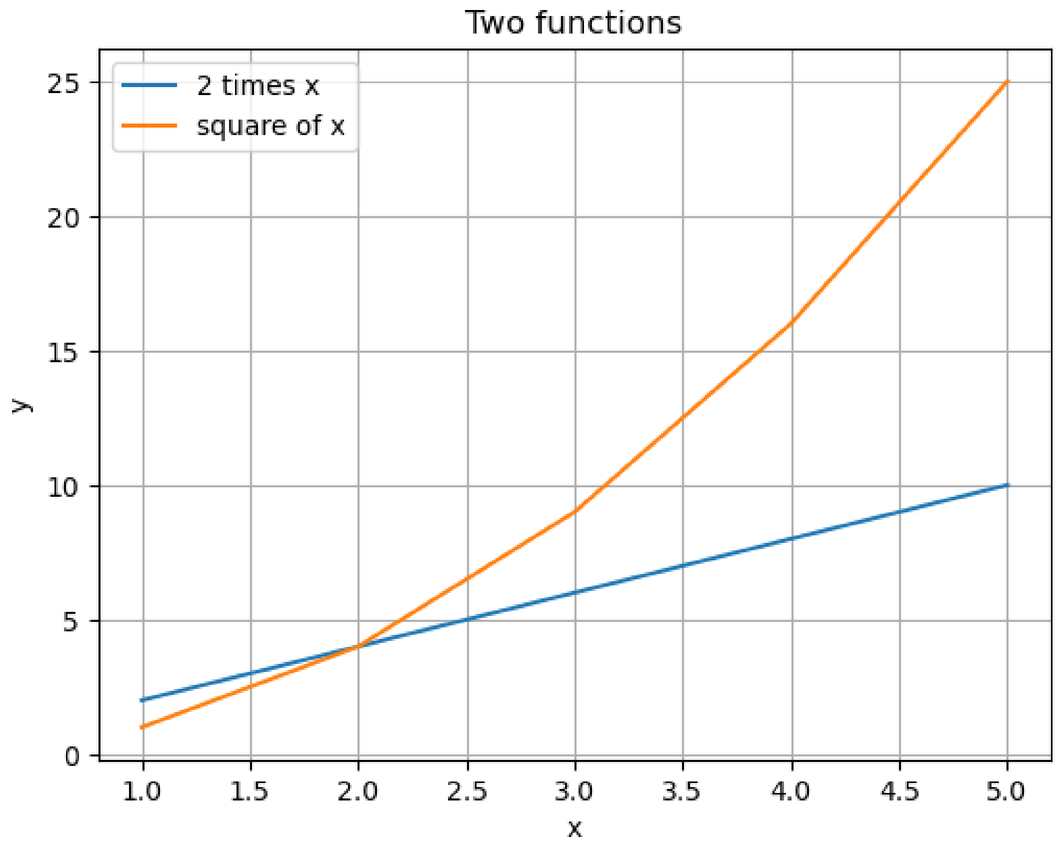
[12]:
```
print(plt.style.available)
```

```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background',
'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright',
'seaborn-colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-
darkgrid', 'seaborn-deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper',
'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-
white', 'seaborn-whitegrid', 'tableau-colorblind10']
```

Let's see a few examples:

[13]:
```
plt.style.use("default")

x1 = [1, 2, 3, 4, 5]
```

```
y1 = [2, 4, 6, 8, 10]
plt.plot(x1, y1, label="2 times x")

x2 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]
plt.plot(x2, y2, label="square of x")

plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Two functions")
plt.grid()

plt.show()
```



## Working with categorical data

Matplotlib can help us visualising categorical and numerical data together by creating **bar charts** and **pie charts** as well. To see how they work, let's borrow an example from one of our earlier lectures.

The table below shows the market shares of major grocery stores in the UK in 2017:

Grocer	Share (%)
Aldi	6.8
Asda	15.7
Lidl	4.9
Morrisons	10.4
Sainsbury's	16.1
Tesco	27.6
The Cooperative	6.1
Waitrose	5.1
Other	7.3

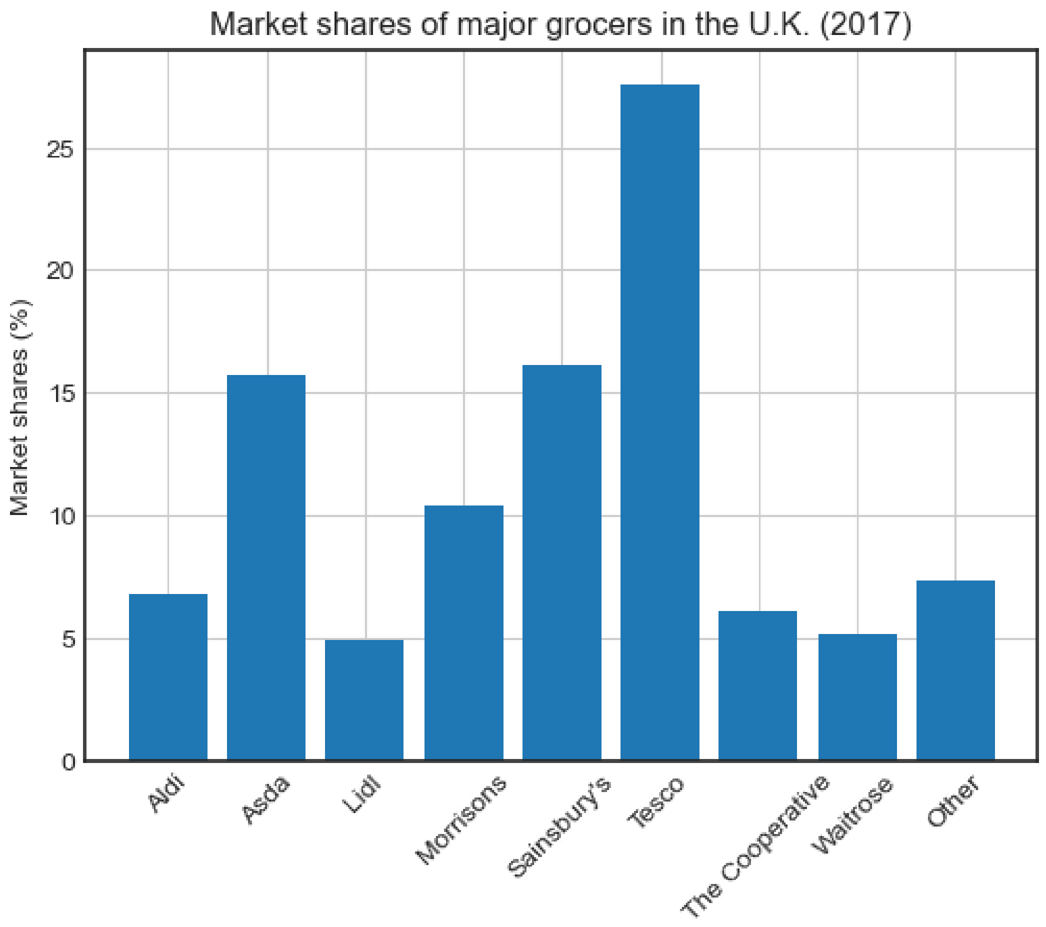
This can be easily organised in a dictionary of lists as follows

[14]:
```
grocery_data = {
    "grocer": ["Aldi", "Asda", "Lidl", "Morrisons", "Sainsbury's", "Tesco",␣
 ↪"The Cooperative", "Waitrose", "Other"],
    "market share": [6.8, 15.7, 4.9, 10.4, 16.1, 27.6, 6.1, 5.1, 7.3]
}
```

To create a bar chart, we will use the `plt.bar()` function.

Similarly to the `plt.plot()` function, we need to pass the values that we would like on the horizontal axis first, and the values on the vertical axis second. In this case, the grocery store names will be passed as x values, and the market share as y values.

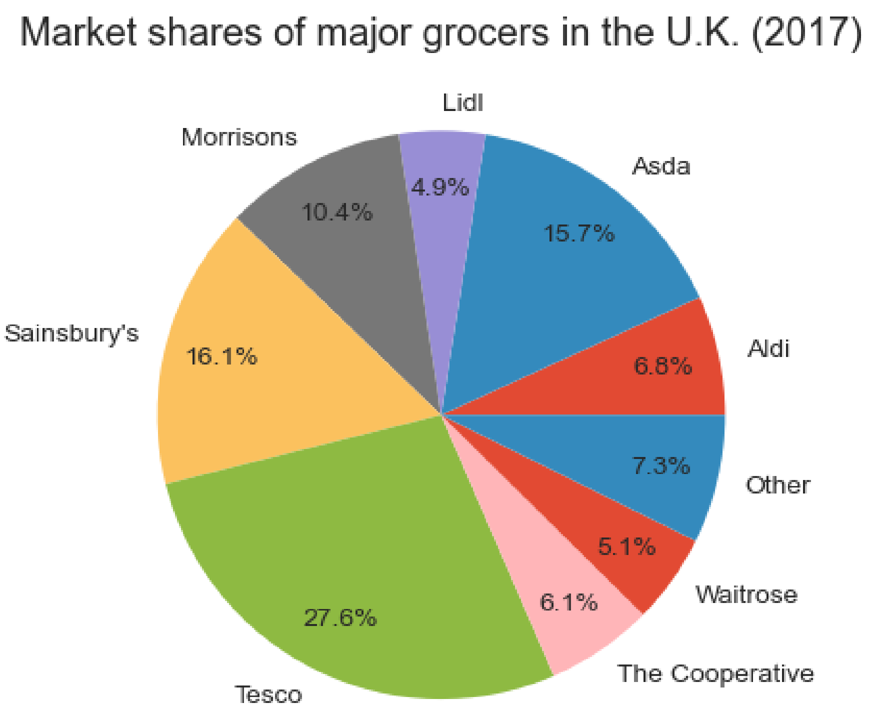
[15]:
```
# Change plotting style
plt.style.use("seaborn-white")

# Use the function ```plt.bar()``` to create a bar chart. Grocer names as x␣
 ↪values, and market shares as y values
plt.bar(grocery_data["grocer"], grocery_data["market share"])

# Rotate x labels to increase reada readability
plt.xticks(rotation=45)

# Add useful information
plt.ylabel("Market shares (%)")
plt.title("Market shares of major grocers in the U.K. (2017)")
plt.grid()

plt.show()
```



And to create a **pie chart** we can use the `plt.pie()` function.

This time the market share values should be passed in as the first argument, and the grocery stores names should be passed using the keyword `label`.

```
[16]: # Change plotting style
plt.style.use("ggplot")

# Use plt.pie() to create a pie chart
plt.pie(grocery_data["market share"], labels=grocery_data["grocer"],␣
 ↪autopct='%1.1f%%', pctdistance=0.8)

plt.title("Market shares of major grocers in the U.K. (2017)")
plt.show()
```



## Additional features and options

### Plotting labelled data

Source: matplotlib official documentation

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in x and y, you can provide the object in the data parameter and just give the labels for x and y:

```
plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

### Displaying figures outside the Jupyter notebook

You have the option of displaying figures outside the notebook with the `%matplotlib` magic command. A **magic** command is a command preceded by a percentage sign `%` that alters the behaviour of Jupyter notebooks and/or controls other settings.

In the case of Matplotlib, we can choose to display our plot in a **separate stand-alone window** with

```
[17]: %matplotlib
```

```
Using matplotlib backend: MacOSX
```

After the magic command is invoked, all plots will displayed in a separate interactive window. This feature is really useful if we want to have a closer look at the plot, and do other things such as zooming.

If we now plot the time-series of unemployment, this will create a figure in a separate window:

[18]:
```
# Sequentially plot unemployment rate on the same figure
for region, unemp_rate in unemployment.items():
    plt.plot(years, unemp_rate, label=region)

# Display useful information
plt.title("U.S. unemployment rate by region")
plt.xlabel("Years")
plt.ylabel("Unemployment rate (%)")
plt.grid()
plt.legend()
plt.show()
```

When working with figures in stand-alone windows outside Jupyter, we need to be careful not to overwrite the current plot. This is because the current figure is still "active", so any plot command will be displayed in the **current** figure!

If, by mistake, I try to create a new plot this will be positioned on top of the current figure.

[19]:
```
# Plot a straigth line
plt.plot(years, [i for i in range(1, len(years)+1)])
plt.show()
```

In order to avoid this, we have two options:

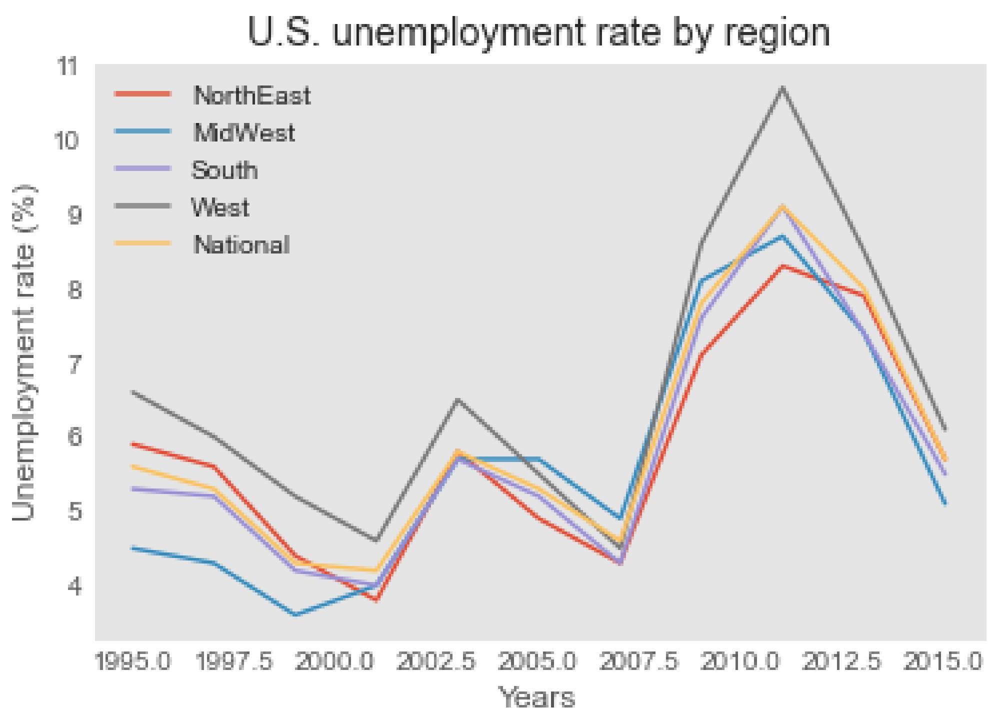
1. Close the current figure before creating a new plot. This is done with `plt.close()`.
2. Create a **new** figure with `plt.figure()`

Either way, `plt.close()` or `plt.figure()` must be invoked **before** the new figure is created.

If you want to revert back to the default behaviour of displaying plots within the Jupyter notebook, you can do so with the following magic command:

[20]:
```
%matplotlib inline
```

[21]:
```
# Sequentially plot unemployment rate on the same figure
for region, unemp_rate in unemployment.items():
    plt.plot(years, unemp_rate, label=region)

# Display useful information
plt.title("U.S. unemployment rate by region")
plt.xlabel("Years")
plt.ylabel("Unemployment rate (%)")
plt.grid()
plt.legend()
plt.show()
```


U.S. unemployment rate by region
NorthEast
MidWest
South
West
National
Unemployment rate (%)
Years
11
10
9
8
7
6
5
4
1995.0
1997.5
2000.0
2002.5
2005.0
2007.5
2010.0
2012.5
2015.0